

6.1903

C Strings And Structs

Fall 2026

Topics

Arrays Review ←

sizeof

C-strings

string.h

structs

structs with functions

Arrays

In C, arrays are just continuous portions of memory accessible via a pointer-like element

```
int x[] = {1,2,3,4}; //make four long int array call it x
int y[10]; //create ten long int array, call it y (But don't declare values)
int z[5] = {1}; //create array 5 long called z, 0th element is 1, rest are 0
```

Can do same thing for char array, but syntax a bit more flexible since we use them for strings:

```
char a[10] = {'t','h','e',' ','c','a','t','.'}; //10 char array with two nulls at end
char b[] = "the cat."; //9 long array with "the cat." followed by null
```

Consider this Code

```
void app_main(){
    int x[] = {1,2,3,4,5}; //compiler figures out array length large enough for items listed
    //10 long array of ints accessible through ptr y
    //only five vals specified (rest are unknown)
    int y[10] = {10,11,12,13};
    printf("%d\n", x[1]); //prints 2
    printf("%d\n", *x); //prints 1 same as x[0] or *(x+0)
    printf("%d\n", x); //prints 1070153528 (0x3fc93f38 in hex)
    printf("%d\n", x+1); //prints 1070153532 (0x3fc93f3c in hex)
    printf("%d\n", *(x+2)); //prints 3...same as x[2]
    printf("%d\n", *(x+5)); //prints undefined...same as x[5]...no idea what's in that memory
    printf("%d\n", *(y+10)); //prints 1...because you bled into where x is and x[0] is 1
}
```

Address:	Value:
0x3fc93f0c:	0x420001ec
0x3fc93f10:	0x0000000a
0x3fc93f14:	0x0000000b
0x3fc93f18:	0x0000000c
0x3fc93f1c:	0x0000000d
0x3fc93f20:	0x00000000
0x3fc93f24:	0x00000000
0x3fc93f28:	0x00000000
0x3fc93f2c:	0x00000000
0x3fc93f30:	0x00000000
0x3fc93f34:	0x00000000
0x3fc93f38:	0x00000001
0x3fc93f3c:	0x00000002
0x3fc93f40:	0x00000003
0x3fc93f44:	0x00000004
0x3fc93f48:	0x00000005

"int y[10]"

"int x[]"

Consider this Code

```
void app_main(){
    int x[] = {1,2,3,4,5}; //compiler figures out array length large enough for items listed
    //10 long array of ints accessible through ptr y
    //only five vals specified (rest are unknown)
    int y[10] = {10,11,12,13};
    printf("%d\n", x[1]); //prints 2
    printf("%d\n", *x); //prints 1 same as x[0] or *(x+0)
    printf("%d\n", x); //prints 1070153528 (0x3fc93f38 in hex)
    printf("%d\n", x+1); //prints 1070153532 (0x3fc93f3c in hex)
    printf("%d\n", *(x+2)); //prints 3...same as x[2]
    printf("%d\n", *(x+5)); //prints undefined...same as x[5]...no idea what's in that memory
    printf("%d\n", *(y+10)); //prints 1...because you bled into where x is and x[0] is 1
}
```

What does the following print?

```
printf("%x\n", *(y+2));
```

But you have to think about this as a Computer experiences it

It doesn't "see" the whole memory ever. It just gets glimpses of it.

If you just tell something that there's an array at 0x3fc93f10 that's great, but there's nothing in memory that tells the program how much memory is associated with that *array*...just where it starts.

Out-of-Bounds For Fun and Profit

You can use this to your advantage.

Where do we get these pictures of memory from?

You have enough knowledge of C to generate them!

Snapshot of Memory

```
#include <stdint.h>
#include <stdio.h>

void print_stack() {
    uint32_t a[] = { 0 };
    printf("address:      value:\n");
    for (int i = 0; i < 20; ++i) {
        printf("%p 0x%.8x\n", a + i, a[i]);
    }
}

int main() {
    uint32_t x = 0xDEADF00D;
    int y[] = { 16, 32, 48, 64, 80, 96 };
    print_stack();
    return 0;
}
```

* Ran this on my home computer; 64-bit word size.

address:	value:
0x7fff9282c5d4	0x00000000
0x7fff9282c5d8	0xcdd42100
0x7fff9282c5dc	0x108074a6
0x7fff9282c5e0	0x9282c620
0x7fff9282c5e4	0x00007fff
0x7fff9282c5e8	0xcbc34271
0x7fff9282c5ec	0x00005dcf
0x7fff9282c5f0	0x00000000
0x7fff9282c5f4	0x00000000
0x7fff9282c5f8	0x00000000
0x7fff9282c5fc	0xdeadf00d
0x7fff9282c600	0x00000010
0x7fff9282c604	0x00000020
0x7fff9282c608	0x00000030
0x7fff9282c60c	0x00000040
0x7fff9282c610	0x00000050
0x7fff9282c614	0x00000060
0x7fff9282c618	0xcdd42100
0x7fff9282c61c	0x108074a6
0x7fff9282c620	0x9282c6c0

Topics

Arrays Review

sizeof 

C-strings

string.h

structs

structs with functions

How to Determine Length of Array?

There is one thing that can kind of help (a little)

sizeof provides the size of a data object

sizeof can sort of help but isn't what you think it is.

```
char a[10] = {'t','h','e',' ','c','a','t','.'}; //10 char array with two nulls
char b[] = "the cat"; //8 long array with "the cat" followed by null
int x;
char y;
int z[15];
printf("%d\n", sizeof(a)); //prints 10
printf("%d\n", sizeof(b)); //prints 8
printf("%d\n", sizeof(x)); //prints 4
printf("%d\n", sizeof(y)); //prints 1
printf("%d\n", sizeof(z)); //prints 60
printf("%d\n", sizeof(z)/sizeof(int)); //prints 15
```

seriously...**sizeof** likely isn't what you think it is. It isn't like `len()` in Python.

"In Functions, Arrays Devolve to Pointers"

In C, all function arguments are **Passed by Value!**

Passing an array in its entirety is expensive*.
Instead we pass a pointer that contains the array's
address (by value) and therefore implicitly provide a
way to pass the array by reference

When an array is handed in as an argument to a
function in C, within the scope of the function, all
that can be seen is the pointer. **NOT THE ARRAY!**

*in time and memory

Implications of Previous Slide

sizeof can still be used in a function, but it will not provide the size of an array pointed to by pointer!

```
int sum_array(int* ai){
    int sum = 0;
    for (int i = 0; i < sizeof(ai); i++){
        sum += ai[i];
    }
    return sum;
}

int main(void){
    int aa[10] = {2,4,6,8,10,12,14,16,18,20};
    int c = sum_array(aa);
    // value of c?
}
```

WHY?

sizeof provides the number of bytes that a *thing* is.
On previous slide, the "*thing*" is a pointer inside a function and that is:

- only four bytes (on 32-bit system)
- or eight bytes (on a 64-bit system)

There is NO WAY to inherently know the size of an array that is passed in as an argument in C.

You MUST:

- Pass along variable informing of size of array
(general solution)

- Establish some sort of rule-set for the size of the object
(C-strings)

Scoping Can Confuse Here

sizeof works on things that are in scope

```
int aa[10] = {2,4,6,8,10,12,14,16,18,20};
```

```
int sum_array(int* ai){  
    int sum = 0;  
    printf("size of ai: %d\n",sizeof(ai));  
    printf("size of aa: %d\n",sizeof(aa));  
    for (int i = 0; i < sizeof(ai); i++){  
        sum+= ai[i];  
    }  
    return sum;  
}
```

```
int main(void){  
    int c = sum_array(aa);  
    printf("%d\n",c);  
}
```

*What is
printed
here?*

sizeof

In original C, sizeof is **NOT a function**, it is an **operator**.

This means it is interpreted at *compile time* rather than *run time*

Before the code even runs, sizeof evaluates all the places where it is used and is replaced with a number

So Code from two pages ago really looks like:

sizeof gets replaced at compile!

```
int aa[10] = {2,4,6,8,10,12,14,16,18,20};

int sum_array(int* ai){
    int sum = 0;
    printf("size of ai: %d\n",      4    );
    printf("size of aa: %d\n",    40    );
    for (int i = 0; i <      4      ; i++){
        sum+= ai[i];
    }
    return sum;
}

int main(void){
    int c = sum_array(aa);
    printf("%d\n",c);
}
```

Conclusion (to be annoying)

Mostly don't use/rely on sizeof for array things in functions.

There is **NO WAY** to inherently know the size of an array that is passed in as an argument in C.

You MUST:

- Pass along variable informing of size of array
(general solution)

- Establish some sort of rule-set for the size of the object
(C-strings)

An array has NO metadata about its size that is accessible at run-time

So if you wanted

More acceptable

```
int sum_array(int* ai, int arr_len){
    int sum = 0;
    printf("size of ai: %d\n", sizeof(ai));
    for (int i = 0; i < arr_len; i++){
        sum+= ai[i];
    }
    return sum;
}

int main(void){
    int aa[10] = {2,4,6,8,10,12,14,16,18,20};
    int bb[5] = {19,20,21,4,1};
    int c = sum_array(aa, sizeof(aa)/sizeof(int));
    int d = sum_array(bb, sizeof(bb)/sizeof(int));
    printf("%d\n", c);
    printf("%d\n", d);
}
```

Conclusion (to be annoying)

Mostly Don't use/rely on sizeof for array things!

There is **NO WAY** to inherently know the size of an array that is passed in as an argument in C.

You MUST:

Pass along variable informing of size of array

(general solution)

solution on previous slide

Establish some sort of rule-set for the size of the object

(C-strings)

solution on upcoming slides

An array has NO metadata about its size that is accessible at run-time

Topics

Arrays Review

sizeof

C-strings 

string.h

structs

structs with functions

A "C-string"

A NULL-terminated Char array

All elements are ASCII characters

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

A “C-string”

A NULL-terminated Char array

Arrays are handed around by pointers

The C-string is from where the pointer points to where
FIRST NULL occurs

```
char a[10] = {'t','h','e',' ','c','a','t','.'}; //10 char array with two nulls at end
```

```
char b[] = "the cat"; //8 long array with "the cat" followed by null
```

This “agreement” about first-NULL allows char arrays to be handed around *without* a supporting length variable in some cases (sacrifice one value in our char encoding space be the “STOP” similar to what we do in DNA/RNA with a stop codon)

Topics

Arrays Review

sizeof

C-strings

string.h



structs

structs with functions

string.h

The string.h library is an important set of functions that allow for convenient c-string manipulation

- Determine attributes of strings

- Manipulate/Modify c-strings

- Compare c-strings

We'll look at a subset of the library.

In Linux or Mac, you can "man <function-name>" for info.

All functions found here:

https://www.tutorialspoint.com/c_standard_library/string_h.htm

strlen

strlen: Determine Length of a C-string

```
int strlen ( const char * str );
```

Takes in pointer to a C-string.

const char * is a way of saying the function won't modify contents of what is pointed to

Returns the length of the string!

"length" is defined as number of non-NULL characters UP UNTIL THE NULL!!!

How would we write this?

strlen

strlen: Determine Length of a C-string

```
int strlen ( const char * str );
```

How would we do this?

```
int custom_strlen ( const char * str ) {  
    const char *searcher = str;  
    while (*searcher != '\0') {  
        searcher++;  
    }  
    return searcher - str;  
}
```

string.h Manipulation Functions

strcpy: copy a c-string

strcat: concatenate a c-string

strcpy

```
char * strcpy ( char * destination, const char * source );
```

Copy to **destination** c-string the contents of c-string pointed to by **source**

Returns pointer to **destination** mainly for convenience

How would this work (what must it do?)

Also...How is this dangerous?

strcpy

```
char * strcpy ( char * destination, const char * source );
```

How might we do this?

```
char * custom_strcpy ( char * destination, const char * source ) {  
    uint32_t i = 0;  
    while (*(source+i) != NULL) {  
        *(destination+i) = source[i];  
        i++;  
    }  
    return destination;  
}
```

strcpy

```
char * strcpy ( char * destination, const char * source );
```

How might we do this?

```
char * custom_strcpy ( char * destination, const char * source ) {  
    uint32_t i = 0;  
    while (*(source+i) != NULL) {  
        *(destination+i) = source[i];  
        i++;  
    }  
    destination[i] = 0; // NULL, '\0' will also work  
    return destination;  
}
```

Danger

Let's run this!

```
int main(void){  
    char prez[] = "Sally Kornbluth";  
    char buffer[4];  
    int len = custom_strlen(prez);  
    printf("length of string: %d\n", len);  
    custom_strcpy(buffer, prez);  
    printf("Original String: \"%s\"\n",prez);  
    printf("Copied String: \"%s\"\n",buffer);  
}
```

length of string: 15
Original String: "y Kornbluth"
Copied String: "Sally Kornbluth"



Danger

prez lives after **buffer** in memory so when we call strcpy, it copies over it!*

```
int main(void){
    char prez[] = "Sally Kornbluth";
    char buffer[4];
    int len = custom_strlen(prez);
    printf("length of string: %d\n", len);
    custom_strcpy(buffer, prez);
    printf("Original String: \"%s\"\n",prez);
    printf("Copied String: \"%s\"\n",buffer);
}
```

length of string: 15

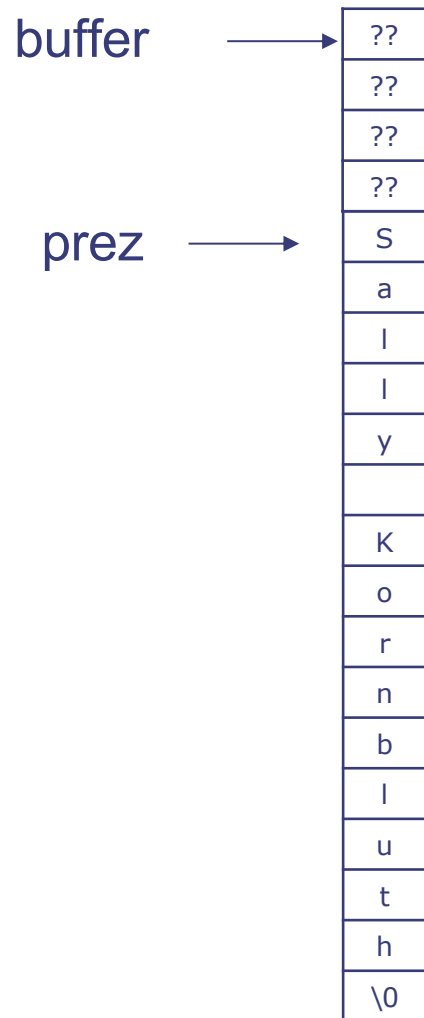
Original String: "y Kornbluth"

Copied String: "Sally Kornbluth"

*real strcpy is no better!

What happened in memory?

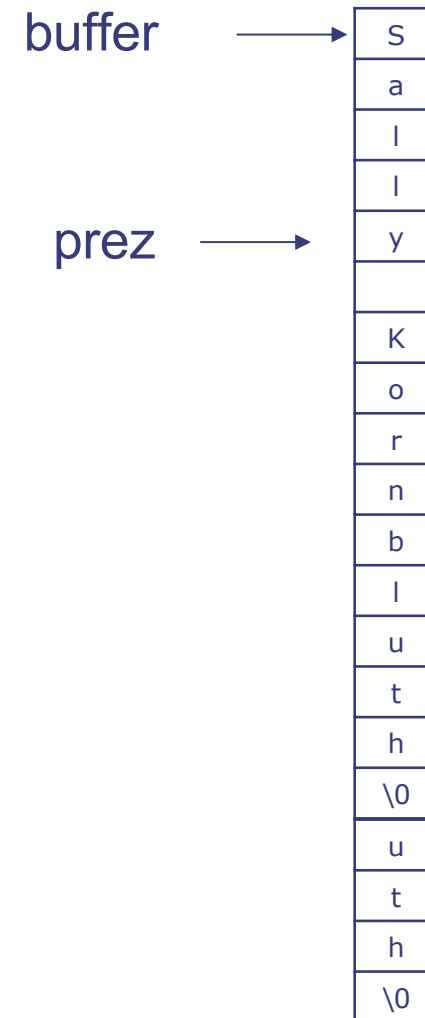
Original Memory:



Then Call:

`strcpy(buffer, prez);`

Final Memory:



How did that actually look in memory?

Original Memory: strcpy(buffer, prez);

Final Memory:

Address:

Value:

0x3fc93f0c:
0x3fc93f10:
0x3fc93f14:
0x3fc93f18:
0x3fc93f1c:
0x3fc93f20:
0x3fc93f24:
0x3fc93f28:
0x3fc93f2c:

?	?	?	?
?	?	?	?
l	l	a	S
o	K		y
l	b	n	r
\0	h	t	u
?	?	?	?
?	?	?	?

buffer

prez

Address:

Value:

0x3fc93f0c:
0x3fc93f10:
0x3fc93f14:
0x3fc93f18:
0x3fc93f1c:
0x3fc93f20:
0x3fc93f24:
0x3fc93f28:
0x3fc93f2c:

?	?	?	?
l	l	a	S
o	K		y
l	b	n	r
\0	h	t	u
\0	h	t	u
?	?	?	?
?	?	?	?

Moral of the story: Check your buffer sizes **before** you strcpy().

strcat

```
char * strcat ( char * destination, const char * source );
```

Add onto **destination** c-string the contents of c-string pointed to by **source**

Returns pointer to **destination** mainly for convenience

How would this work (what must it do?)

How is this dangerous?

strcat

```
char * strcat ( char * destination, const char * source );
```

How might we write this?

```
int custom_strlen ( const char * str );  
char * custom_strcpy ( char * destination, const char * source );
```

strcat

```
char * strcat ( char * destination, const char * source );
```

How might we write this?

```
int custom_strlen ( const char * str );  
char * custom_strcpy ( char * destination, const char * source );  
  
char * custom_strcat ( char * destination, const char * source ) {  
    return custom_strcpy(destination + custom_strlen(destination),  
                           source);  
}
```

string.h Manipulation Functions

strcpy: copy a c-string

strncpy: copy n characters from a c-string

strcat: concatenate a c-string

strncat: concatenate n-characters from a c-string

These are nice since they provide a way to restrict how many letters get written in each case

Use in-scope knowledge about c-string to protect writing outside memory!

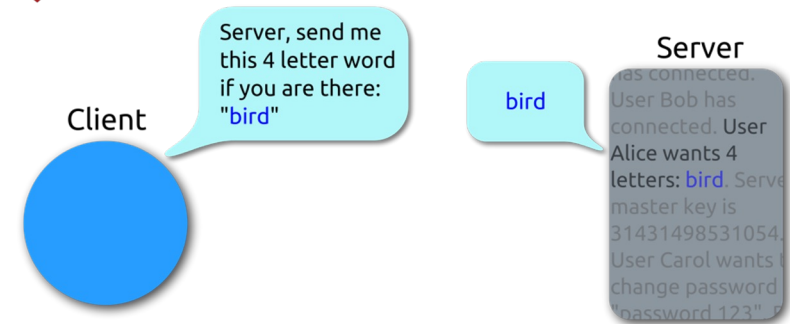
Buffer Overread/Overwrite



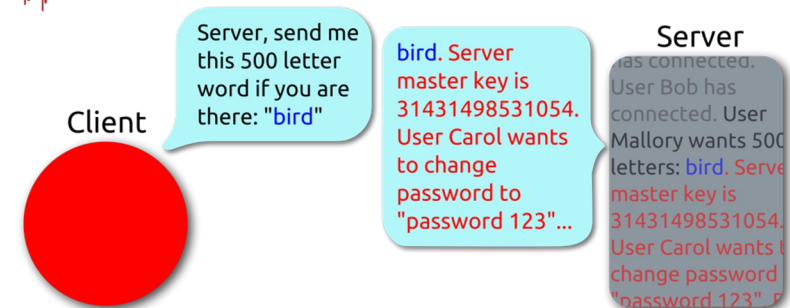
Famous bug announced
in openSSL in 2014
The bug could leak
information by not
respecting buffer
boundaries



Heartbeat – Normal usage



Heartbeat – Malicious usage



<https://en.wikipedia.org/wiki/Heartbleed>

Even safer string.h Functions

strcpy: copy a c-string

strncpy: copy n characters from a c-string

strncpy: copy n-1 characters from a c-string and makes sure to NULL terminate

strcat: concatenate a c-string

strncat: concatenate n characters from a c-string

strlcat: concatenate n-1 characters from a c-string and makes sure to NULL terminate

string.h Comparison Functions

strcmp

```
int strcmp ( const char * str1, const char * str2 );
```

strncmp

```
int strncmp ( const char * str1, const char * str2, size_t n );
```

Both of these compare two c-strings.

strncmp compares the first *n* characters

strcmp compares the entire c-string!

What does “compare” mean? It returns:

<0 if the first character that does not match has a lower value in *str1* than in *str2*

0 if the contents of both strings are equal

>0 if the first character that does not match has a greater value in *str1* than in *str2*

string.h Search Functions

strchr: Find first instance of character in c-string

strrchr: Find last instance of character in c-string

strstr: Find location of sub c-string inside c-string

strtok: Tokenize c-string into sub c-strings.

(exercise this week!)

strchr

```
char * strchr ( const char * haystack, char needle );
```

Looks for first instance of **needle** in c-string **haystack** and returns a pointer to it

If it isn't found, it returns a **NULL** Pointer!

Example:

Determine if there is an 'l' in "Sally Kornbluth"

How could we determine where this first occurrence is?

Aside...Why is NULL ok for a return value?

NULL in C is defined as the value 0.

In general *nothing* is going to be sitting at memory location 0 (which is NULL). So this is a good value to make a pointer to convey the concept of “invalidity” or “failure to find”.

Other languages such as Python have a whole datatype to do this (Python's None type)

strchr

```
char * strchr ( const char * haystack, char needle );
```

How might we do this?

```
char * custom_strchr ( const char * haystack, char needle ) {  
    uint32_t i = 0;  
    while (haystack[i] != NULL) {  
        if (*(haystack+i) == needle) {  
            return haystack + i;  
        }  
        i++;  
    }  
    return NULL;  
}
```

strrchr

```
char * strrchr ( const char * haystack, char needle );
```

Looks for last instance of **needle** in c-string **haystack** and returns a pointer to it

How must this function's search *differ* from regular strchr?

strstr

```
char * strstr ( const char * haystack, const char * needle );
```

Looks for first instance of c-string **needle** in c-string **haystack** and returns a pointer to it beginning of **needle**

Returns NULL if not found!

sprintf?

sprintf is a function included in standard C that is not part of string.h, but is often used to do similar tasks as string.h functions

Just like **printf**, but writes to char buffer rather than

`terminal`
`int sprintf (char * str, const char * format, ... );`

Write to **str** using regular string formatting rules like %d, etc...

Function returns the number of characters written!

Uses for sprintf

Allows the creation of formatted C-strings rather than just printing them to terminal!

Example, I want to stringify a computationally determined number:

```
int i = some_function(19,86); //return 2012
char temp[10];
sprintf(temp,"%d",i);
//temp now holds the c-string: "2012\0"
//( '\0' is auto-added in sprintf; It's in format.)
```

It can also be used to do replicate/provide slight deviations/improvements on several of the string.h functions

Generate C-string that has every number in it from 0 to 1000!

One function that repeatedly uses strcat:

```
//version one using strcat:
char totes[100000]; //big enough
void build_num_string_1(){
    totes[0] = 0; //null first value
    for (int i = 0; i < 1000; i++){
        char temp[10];
        sprintf(temp, "%d ", i);
        strcat(totes, temp);
    }
    printf("%s\n", totes);
}
```

One function that uses sprintf in conjunction with pointer arithmetic directly!:

```
//version 2 using sprintf and
//pointer arithmetic:
char totes[100000]; //big enough
void build_num_string_2(){
    totes[0] = 0; //null first value
    int tally = 0;
    for (int i = 0; i < 1000; i++){
        tally += sprintf(totes+tally, "%d ", i);
    }
    printf("%s\n", totes);
}
```

Results?

```
#include<stdio.h>
#include<string.h>
#include<time.h>

int main(void){
    clock_t start; //like an unsigned int
    clock_t end; //like an unsigned int
    double time_used; //64 bit int
    start = clock(); //store start time
    build_num_string_1();
    end = clock(); //store end time
    time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("num_string_1 took %f seconds\n",time_used);
    start = clock(); //store start time
    build_num_string_2();
    end = clock(); //store end time
    time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("num_string_2 took %f seconds\n",time_used);
}
```

Generate C-string that has every number in it from 0 to 1000!

One function that repeatedly uses strcat:

```
//version one using strcat:
char totes[100000]; //big enough
void build_num_string_1(){
    totes[0] = 0; //null first value
    for (int i = 0; i < 1000; i++){
        char temp[10];
        sprintf(temp, "%d ", i);
        strcat(totes, temp);
    }
    printf("%s\n", totes);
}
```

0.000279 secs

One function that uses sprintf in conjunction with pointer arithmetic directly!:

```
//version 2 using sprintf and
//pointer arithmetic:
char totes[100000]; //big enough
void build_num_string_2(){
    totes[0] = 0; //null first value
    int tally = 0;
    for (int i = 0; i < 1000; i++){
        tally += sprintf(totes+tally, "%d ", i);
    }
    printf("%s\n", totes);
}
```

0.000132 secs

Generate C-string that has every number in it from 0 to 1000!

One function that repeatedly uses strcat:

```
//version one using strcat:
char totes[10000000]; //big enough
void build_num_string_1(){
    totes[0] = 0; //null first value
    for (int i = 0; i < 100000; i++){
        char temp[10];
        sprintf(temp, "%d ", i);
        strcat(totes, temp);
    }
    printf("%s\n", totes);
}
```

0.563427 secs

One function that uses sprintf in conjunction with pointer arithmetic directly!:

```
//version 2 using sprintf and
//pointer arithmetic:
char totes[10000000]; //big enough
void build_num_string_2(){
    totes[0] = 0; //null first value
    int tally = 0;
    for (int i = 0; i < 100000; i++){
        tally += sprintf(totes+tally, "%d ", i);
    }
    printf("%s\n", totes);
}
```

0.003983 secs

Some Conclusions

strcat repeatedly has to scan the string from the beginning, causing longer and longer run-times for longer and longer strings!

Avoid repeatedly scanning a string if you can help it!

The diagram illustrates the lineage of programming languages. Key nodes and their connections include:

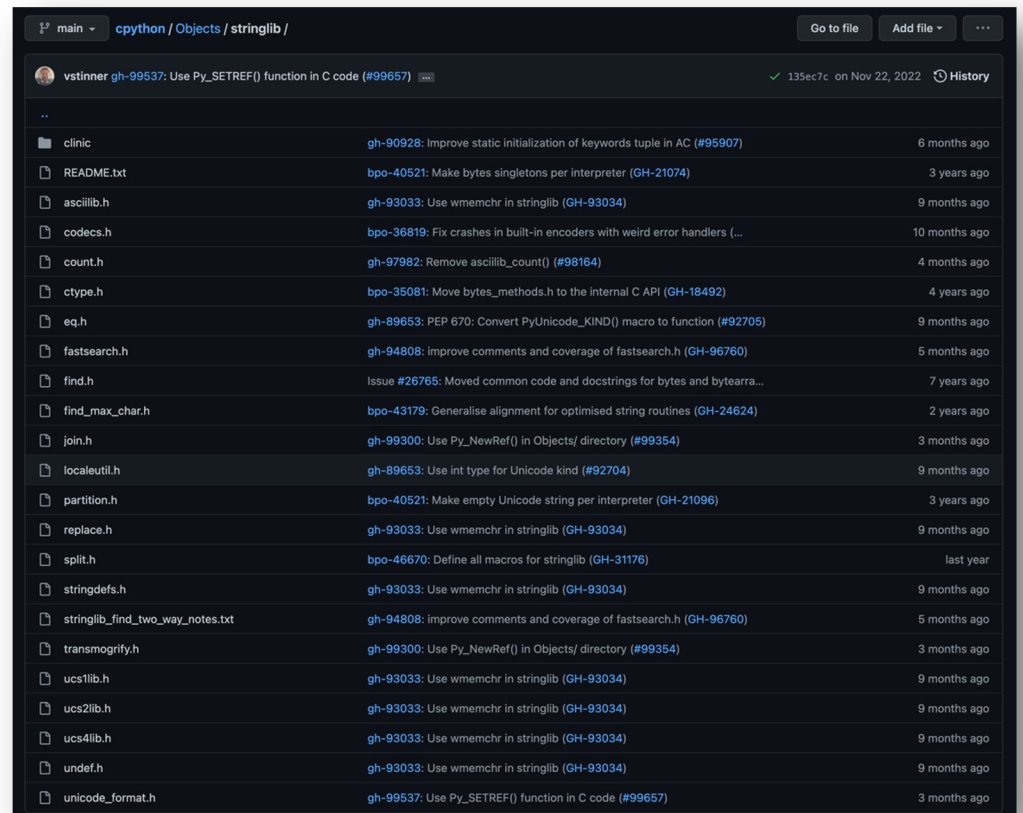
- Fortran I** (1956) influences **Algol 60**, **COBOL**, **Lisp**, and **Fortran 77**.
- Algol 60** (1960) influences **Smalltalk**, **Pascal**, **Scheme**, **C (K&R)**, **Smalltalk 80**, **Python**, **Ruby**, **Perl**, **Ada 83**, **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- COBOL** (1962) influences **PL/I**, which in turn influences **Pascal**.
- Lisp** (1958) influences **Scheme**, **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Smalltalk** (1970) influences **Smalltalk 80**, **Python**, **Ruby**, **Perl**, **Ada 83**, **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Pascal** (1971) influences **Scheme**, **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Scheme** (1975) influences **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Prolog** (1972) influences **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Fortran 77** (1977) influences **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- C (K&R)** (1978) influences **Smalltalk 80**, **Python**, **Ruby**, **Perl**, **Ada 83**, **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Smalltalk 80** (1980) influences **Python**, **Ruby**, **Perl**, **Ada 83**, **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Ada 83** (1983) influences **Common Lisp**, **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Common Lisp** (1984) influences **SML**, **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- SML** (1985) influences **Caml**, **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Caml** (1986) influences **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Fortran 90** (1990) influences **Ocaml**, **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Ocaml** (1996) influences **Haskell 98**, **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Haskell 98** (1998) influences **Haskell 2010**, **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Haskell 2010** (2010) influences **Rust**, **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Rust** (2010) influences **Swift**, **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- Swift** (2014) influences **JavaScript ES2017**, **C# 5.0**, **Kotlin**, **Java 8**, **Java 5**, **C++ (ISO)**, **C#**, **Python 2.0**, **Go**, **C++ (K&R)**, **Tcl**, **JavaScript**, and **Java**.
- JavaScript ES2017** (2017)

So when we += to a python string, that length is already known inside the object

L03-55

Python's String Library

1000's and 1000's of
lines of well written C



cpython / Objects / stringlib /		
..	gh-90928: Improve static initialization of keywords tuple in AC (#95907)	6 months ago
clinic	bpo-40521: Make bytes singletons per interpreter (GH-21074)	3 years ago
README.txt	gh-93033: Use wmemchr in stringlib (GH-93034)	9 months ago
asciilib.h	bpo-36819: Fix crashes in built-in encoders with weird error handlers (...)	10 months ago
codecs.h	gh-97982: Remove asciilib_count() (#98164)	4 months ago
count.h	bpo-35081: Move bytes_methods.h to the internal C API (GH-18492)	4 years ago
ctype.h	gh-89653: PEP 670: Convert PyUnicode_KIND() macro to function (#92705)	9 months ago
eq.h	gh-94808: improve comments and coverage of fastsearch.h (GH-96760)	5 months ago
fastsearch.h	Issue #26765: Moved common code and docstrings for bytes and bytearray...	7 years ago
find.h	bpo-43179: Generalise alignment for optimised string routines (GH-24624)	2 years ago
find_max_char.h	gh-99300: Use Py_NewRef() in Objects/ directory (#99354)	3 months ago
join.h	gh-89653: Use int type for Unicode kind (#92704)	9 months ago
localeutil.h	bpo-40521: Make empty Unicode string per interpreter (GH-21096)	3 years ago
partition.h	gh-93033: Use wmemchr in stringlib (GH-93034)	9 months ago
replace.h	bpo-46670: Define all macros for stringlib (GH-31176)	last year
split.h	gh-93033: Use wmemchr in stringlib (GH-93034)	9 months ago
stringdefs.h	gh-94808: improve comments and coverage of fastsearch.h (GH-96760)	5 months ago
stringlib_find_two_way_notes.txt	gh-99300: Use Py_NewRef() in Objects/ directory (#99354)	3 months ago
transmogrify.h	gh-93033: Use wmemchr in stringlib (GH-93034)	9 months ago
ucs1lib.h	gh-93033: Use wmemchr in stringlib (GH-93034)	9 months ago
ucs2lib.h	gh-93033: Use wmemchr in stringlib (GH-93034)	9 months ago
ucs4lib.h	gh-93033: Use wmemchr in stringlib (GH-93034)	9 months ago
undef.h	gh-93033: Use wmemchr in stringlib (GH-93034)	9 months ago
unicode_format.h	gh-99537: Use Py_SETREF() function in C code (#99657)	3 months ago

<https://github.com/python/cpython/tree/main/Objects/stringlib>

Topics

Arrays Review

sizeof

C-strings

string.h

structs



structs with functions

Structs!

So far have individual variables, and arrays

What if we want a more heterogeneous data object?

Structs to the rescue!

Can have member variables, but do not have member methods or functions

Defining Struct

Name it!

Can have member variables but *does not* have member methods!

Access member variables with dot operator:

(thing.x, for example to access thing's x member)

For example here's a struct that describes a subject

```
struct Subject{  
    int dept;  
    int units;  
    char name[100];  
    int num_students;  
};
```

Example

```
#include<stdio.h>
#include<string.h>
struct Subject{
    int dept;
    int units;
    char name[100];
    int num_students;
};

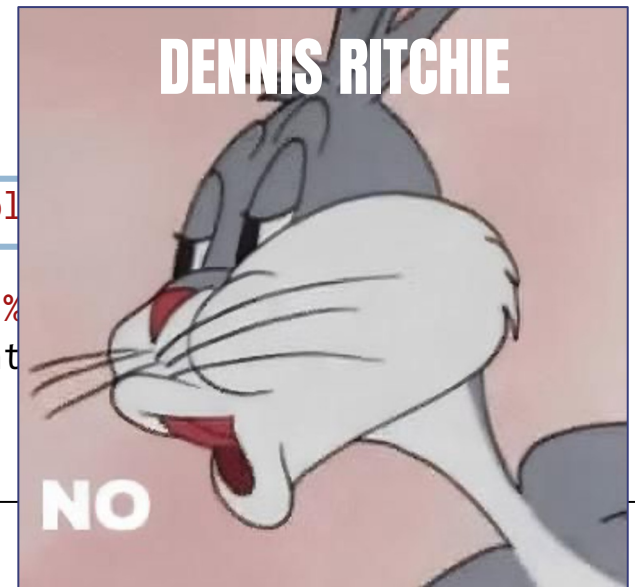
int main(void){
    struct Subject our_course;
    our_course.dept = 6;
    our_course.units = 6;
    strcpy(our_course.name, "6.1903: Intro to C and Assembly");
    our_course.num_students = 255;
    printf("%s has %d students and is in Course %d and is %d credits.\n",
           our_course.name, our_course.num_students,
           our_course.dept, our_course.units);
}
```

Example

```
#include<stdio.h>
#include<string.h>
struct Subject{
    int dept;
    int units;
    char name[100];
    int num_students;
};

int main(void){
    struct Subject our_course;
    our_course.dept = 6;
    our_course.units = 6;
    strcpy(our_course.name, "6.1903: Intro to C and Assembly");
    our_course.num_students = 255;
    printf("%s has %d students and is in Course %d and is %d units",
           our_course.name, our_course.num_students,
           our_course.dept, our_course.units);
}
```

Can we do
our_course.name = "6.1903: Intro to C and Assembly";



Topics

Arrays Review

sizeof

C-strings

string.h

structs

structs with functions



Working with a Struct

Let's make a function:

```
//takes in struct Subject argument
void make_subject1(struct Subject s){
    s.dept = 8;
    s.units = 12;
    strcpy(s.name, "8.02: Physics 2");
    s.num_students = 500;
}
```

Then call it:

```
//assume our_course is still as previously defined!
make_subject1(our_course); //call function on our_course
printf("%s has %d students and is in Course %d and is %d credits.\n",
        our_course.name, our_course.num_students,
        our_course.dept, our_course.units);
```

What will this print?

Arguments Passed BY VALUE!!!!!!

Arguments are always passed by Value! So in previous function we were making copy of struct Subject! Working on it, then forgetting it!

Fix by returning new Struct:

```
//function takes in struct Subject, returns a struct Subject
struct Subject make_subject2(struct Subject s){
    s.dept = 8;
    s.units = 12;
    strcpy(s.name, "8.02: Physics 2");
    s.num_students = 500;
    return s;
}
```

BY VALUE!!!!

Try this:

```
//takes in struct Subject, returns a struct Subject
struct Subject make_subject2(struct Subject s){
    s.dept = 8;
    s.units = 12;
    strcpy(s.name, "8.02: Physics 2");
    s.num_students = 500;
    return s;
}
```

```
our_course = make_subject2(our_course); //call function on our_course return/overwrite
printf("%s has %d students and is in Course %d and is %d credits.\n",
        our_course.name, our_course.num_students,
        our_course.dept, our_course.units);
```

What will this print?

Is Passing a Struct a Good Idea?

In C, everything is passed by value.

To pass by value means a copy needs to be made
Structs can be big (just like arrays), so passing a large thing by value can be:

- Slow

- Use limited memory to make temporary copy

 - Use too much and you could get a stack overflow! (final lecture)

 - Could be catastrophic on embedded system or wasteful on a performant system

So passing a struct by value, while allowed, is often not the best idea.

Struct pointer?

Just like everything else, you can have a pointer to a struct

The pointer allows us to reference things in memory (round-about way of passing by reference) and modify it directly!

```
void make_subject3(struct Subject *s);
```

```
make_subject3(&our_course); //call function on pointer to our_course
printf("%s has %d students and is in course %d and is %d credits.\n",
       our_course.name, our_course.num_students,
       our_course.dept, our_course.units);
```

What would/should this print?

Struct pointer!

```
void make_subject3 (struct Subject* s) ;
```

How to access members of struct pointer?

*s.units = 12; ???

NO * has lower priority than .

So this is doing *(s.units)=12;

So how about (*s).units = 12;???

YES, but that's ugly

Preferred is s->units = 12;

Combo of * + .

Why does C have two distinct operators? Because shut up. That's how Ritchie "designed" it!!!!1!

C Operator Hierarchy

Notice how `.` is higher than `*`

Appendix 3: C Operator Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	? :	Ternary conditional	Right-to-left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= =	Assignment by bitwise AND, XOR, and OR	

Working with Struct Pointer!

So putting it all together:

```
void make_subject3(struct Subject* s){  
    s->dept = 8; //or (*s).dept  
    s->units = 12; //or (*s).units  
    strcpy((*s).name, "8.02: Physics 2"); //or s->name  
    (*s).num_students = 500; //or s->num_students  
}
```

```
make_subject3(&our_course); //call function on pointer to our_course  
printf("%s has %d students and is in Course %d and is %d credits.\n",  
        our_course.name, our_course.num_students,  
        our_course.dept, our_course.units);
```

8.02: Physics 2 has 500 students and is in Course 8 and is 12 credits.

How big is a Struct?

At least* as big as the sum of its component parts

```
struct Subject{  
    int dept;  
    int units;  
    char name[100];  
    int num_students;  
};
```

A Subject struct would therefore be (on a 32 bit

*possibly more if words need to be aligned

Couple other struct things:

No member functions, so no concept of public/private

Struct can have member structs...just not member of same type (Subject cannot have member Subject)

Can declare structs with starting values:

[https://en.wikipedia.org/wiki/Struct_\(C_programming_language\)](https://en.wikipedia.org/wiki/Struct_(C_programming_language))

Initializing Structs

Can define structs at creation too like this:

```
//declare but don't initialize:
struct Subject our_course;
//declare and initialize:
struct Subject other_course = {9, 12, "9.01: Intro Brain Stuff", 100};
//declare and initialize (named fields):
struct Subject other_other_course = {.dept=1, .units=12,
                                     .name="1.01: Intro Prob/Inference",
                                     .num_students=100};
```

The Absence of Anything... "None"

C just says "NULL" is nothing.

Python defines a struct called `_PyNone_Type`

Line 2017 of `object.c` inside the `objects` folder of Cpython distribution

```
PyTypeObject _PyNone_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "NoneType",
    0,
    0,
    none_dealloc, /*tp_dealloc*/
    0, /*tp_vectorcall_offset*/
    0, /*tp_getattr*/
    0, /*tp_setattr*/
    0, /*tp_as_async*/
    none_repr, /*tp_repr*/
    &none_as_number, /*tp_as_number*/
    0, /*tp_as_sequence*/
    0, /*tp_as_mapping*/
    (hashfunc)none_hash, /*tp_hash */
    0, /*tp_call */
    0, /*tp_str */
    0, /*tp_getattro */
    0, /*tp_setattro */
    0, /*tp_as_buffer */
    Py_TPFLAGS_DEFAULT, /*tp_flags */
    0, /*tp_doc */
    0, /*tp_traverse */
    0, /*tp_clear */
    _Py_BaseObject_RichCompare, /*tp_richcompare */
    0, /*tp_weaklistoffset */
    0, /*tp_iter */
    0, /*tp_iternext */
    0, /*tp_methods */
    0, /*tp_members */
    0, /*tp_getset */
    0, /*tp_base */
    0, /*tp_dict */
    0, /*tp_descr_get */
    0, /*tp_descr_set */
    0, /*tp_dictoffset */
    0, /*tp_init */
    0, /*tp_alloc */
    none_new, /*tp_new */
};
```